Measuring the Regularity of Array References [*]

Erin Parker[†] Bronis R. de Supinski[‡] Daniel J. Quinlan[‡]
parker@cs.unc.edu bronis@llnl.gov dquinlan@llnl.gov
[†]Department of Computer Science The University of North Carolina Chapel Hill, NC 27599-3175
[‡]Lawrence Livermore National Laboratory Center for Applied Scientific Computing Livermore, CA 94551

1. INTRODUCTION

The running times of large scientific programs are strongly influenced by the time spent accessing main memory. Many mechanisms, such as prefetching [3], exploit regular access patterns in order to overlap memory accesses with computation and, thus, reduce memory stall cycles. The benefit of these mechanisms depends on the regularity of an application's memory accesses. Although several access descriptors have been proposed [2, 4], access regularity is an intuitive concept for which few formal metrics exist [1].

We consider a program to be *regular* if it contains array references with identifiable access patterns that are repeated as memory is traversed. For our purposes, we restrict this definition to linear patterns. We present a set of metrics that quantify access regularity. We have implemented a source-to-source compiler mechanism to measure access regularity. Results on our sample code demonstrate that our analysis mechanism is fast and accurate.

2. THREE APPROACHES

We present three approaches for measuring the regularity of a program. Our static approach is a low run-time overhead mechanism that uses statically-determined information, augmented at run time only by simple scalar data, such as loop bounds. Our dynamic approach instruments array references so that their regularity can be precisely determined at run time; this approach has significant run-time overhead but is highly accurate. Our overall approach is a hybrid of the static and dynamic approaches. It provides high accuracy with reasonable run-time overhead by using statically-determined information where possible.

The *static approach* examines a program's AST (Abstract Syntax Tree) at compile time to gather knowledge of its loop nests and the array references made within the loop nests. Based on analysis of the array index expressions, we categorize an array reference as regular, irregular or indeterminate. A *regular* array reference is one in which all indices are linear expressions of the LCVs (Loop Control Variables). An *irregular* array reference is one in which at least one index is a nonlinear expression of the LCVs. An array reference is *indeterminate* if at least one index is an expression that cannot be analyzed at compile time or the array reference is contained in the body of a conditional. For example, the array reference A[B[i]] is indeterminate without knowledge of how array B is initialized, and the array reference A[f(i)] is indeterminate without knowledge of what is returned by the function f given input i. Although more aggressive compile-time analysis can categorize some occurrences of these two examples as regular or irregular array references, in general, their regularity cannot be determined until run time.

For any regular array reference, each execution of the innermost loop enclosing it will generate a predictable stream of array accesses. We call such a stream a *regular stream*. Our linear restriction implies that array references we classify as irregular do not constitute a regular stream. Indeterminate array references may be irregular; our static approach assumes that they are. Therefore, based on analysis of the LCVs of the loop nests containing regular array references, we can compute the number of regular streams, their average length, and the proportion of array accesses that occur in regular streams, among other statistics.

The *dynamic approach* examines a program's AST to locate array references contained in loop nests. It does not analyze the indices of array references or LCVs of loop nests. Instead, we instrument the AST with instructions for tracking the actual value of the index to an array reference. A stream of indices form a regular stream if the stride between all values is the same. We keep the same statistics for regular streams as in the static approach. The dynamic approach accurately categorizes all array references although it makes no attempt to categorize them statically.

The *hybrid approach* combines the two approaches described above. As in the static approach, we categorize an array reference as regular, irregular or indeterminate. Then for every regular array reference, we compute the statistics for its regular streams. However, instead of conservatively assuming that every indeterminate array reference is irregular, we perform run-time tracking of array indices to discern actual regularity, as in the dynamic approach.

It is clear that the static approach incurs virtually no run-time overhead, but its accuracy can vary widely and is based on the number of indeterminate array references in a program. The dynamic approach enjoys great accuracy at the cost of a noticeable run-time overhead. The hybrid approach is designed to incur larger run-time overhead only when it is necessary for greater accuracy. This relationship among the accuracy and overhead of the three approaches is demon-

[*]Work performed under the auspices of the U. S. Department of Energy by University of California Lawrence Livermore National Laboratory under Contract W-7405-Eng-48, UCRL-JC-144545-abs. Erin Parker is supported by a DOE High-Performance Computer Science Fellowship.

strated in Section 3.

We accomplish automatic analysis and instrumentation of the AST using ROSE [5]. ROSE is a tool for building source-to-source preprocessors. The preprocessor generates an AST from the program source code; the AST is then used for analysis, instrumentation or optimization. The instrumentation of our static approach merely computes the regularity statistics once the values of any run-time constants are known. In our dynamic approach, our instrumentation actually tracks array index values and detects any regularity. The hybrid approach only uses the more expensive run-time instrumentation for indeterminate references .

3. RESULTS

In this section, we discuss the accuracy and run-time overhead of our three approaches for a simple test program. This example program clearly demonstrates the trade-offs between our approaches; the poster will include results for less trivial programs.

```
do i = 0, regularity_param
  do j = 0, MAX
    sum += A[j]

do i = 0, 100 - regularity_param
  do j = 0, MAX
    sum += A[B[j]]
```

Note that *regularity_param* is an integer provided by the user at run time whose value is between 0 and 100. B is an array of integers with size at least MAX, which has been initialized in one of two ways. In Case 1, B[i] is a random integer with a value between 0 and MAX-1, and in Case 2, B[i] = i.

Our example program has three array access streams: the accesses to the A array in both loops and the accesses to the B array that determine the A indices in the second loop. Therefore, in Case1, $100/(200 - \textit{regularity\_param})$% of array accesses occur in regular streams, and, in Case 2, 100% of array accesses occur in regular streams. All three of our approaches correctly detect regularity in Case 1 of the sample program. However, for Case 2 of the sample program, our static approach misclassifies array reference A[B[j]] as irregular, while our dynamic and hybrid approaches correctly classify it as regular.

In Figure 1, we see that the running time required by the source code instrumented using our hybrid approach is proportional to the number of indeterminate array references that must be tracked at run time, as expected. Notice that the run-time overhead of our hybrid approach is significantly less then that of our dynamic approach even when the value of *regularity_param* is 0. Although the indeterminate array reference A[B[j]] must be instrumented for run-time detection of regularity, our hybrid approach saves run-time overhead by statically categorizing the array reference B[j] as regular.

4. FUTURE WORK

The effort to measure regularity in programs is ongoing, and the preliminary work discussed here has raised several issues. It is undesirable to use our dynamic approach to measure the regularity of large LLNL codes, as it will add overhead to already long-running programs. Likewise, the possible inaccuracy of our static approach on complicated programs makes it unsuitable. Therefore, we are interested to see the accuracy/overhead trade-off of using our hybrid approach on such programs. Furthermore, our analysis can be expanded to include references to array class objects in use at LLNL, which we expect to introduce new challenges.

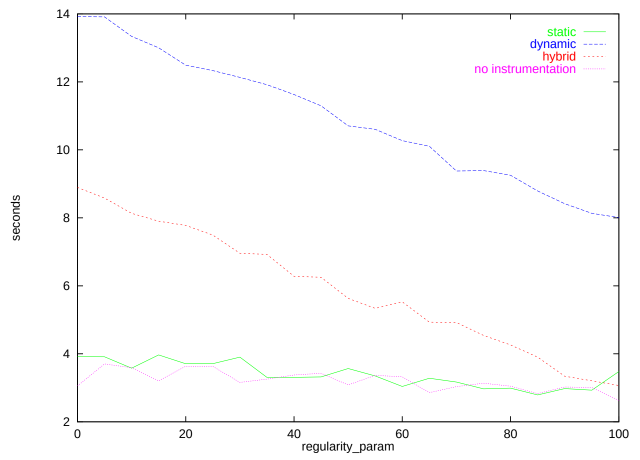

Figure 1: Running time of the example program when instrumented according to each of the three approaches (MAX=100,000).

5. REFERENCES

[1] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of ACM PLDI*, pages 191–202, June 2001.

[2] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[3] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998.

[4] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of ACM PLDI*, volume 33, pages 60–71, May 1998.

[5] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. In *Parallel Processing Letters*, volume 10, 2000. Also presented at Conference on Parallel Compilers (CPC2000), Aussois, France, January 2000.